Teaching Programming with Computational and Informational Thinking

Greg Michaelson, School of Mathematical and Computer Sciences, Heriot-Watt University
Contact: G.Michaelson@hw.ac.uk

Keywords: computational thinking; informational thinking; programming; teaching.

1. Background

Computers are the dominant technology of the early 21st century: pretty well all aspects of economic, social and personal life are now unthinkable without them. In turn, computer hardware is controlled by software, that is, codes written in programming languages. Programming, the construction of software, is thus a fundamental activity, in which millions of people are engaged worldwide, and the teaching of programming is long established in international secondary and higher education. Yet, going on 70 years after the first computers were built, there is no well-established pedagogy for teaching programming.

There has certainly been no shortage of approaches. However, these have often been driven by fashion, an enthusiastic amateurism or a wish to follow best industrial practice, which, while appropriate for mature professionals, is poorly suited to novice programmers. Much of the difficulty lies in the very close relationship between problem solving and programming. Once a problem is well characterised it is relatively straightforward to realise a solution in software. However, teaching problem solving is, if anything, less well understood than teaching programming.

Problem solving seems to be a creative, holistic, dialectical, multi-dimensional, iterative process. While there are well established techniques for analysing problems, arbitrary problems cannot be solved by rote, by mechanically applying techniques in some prescribed linear order. Furthermore, historically, approaches to teaching programming have failed to account for this complexity in problem solving, focusing strongly on programming itself and, if at all, only partially and superficially exploring problem solving.

Recently, an integrated approach to problem solving and programming called *Computational Thinking* (CT) (Wing, 2006) has gained considerable currency. CT has the enormous advantage over prior approaches of strongly emphasising problem solving and of making explicit core techniques. Nonetheless, there is still a tendency to view CT as prescriptive rather than creative, engendering scholastic arguments about the nature and status of CT techniques. Programming at heart is concerned with processing information but many accounts of CT emphasise processing over information rather than seeing then as intimately related.

In this paper, while acknowledging and building on the strengths of CT, I argue that understanding the form and structure of information should be primary in any pedagogy of programming.

2. Computational Thinking, Computer Science and Computing

2.1. Overview

Computing involves making models of the world, to understand it and change it in predictable ways. We can then write programs that map the abstractions of a conceptual model onto the concrete behaviours of a physical computer so the behaviour of the program on the computer tells us about how the world works.

We make models by solving real world problems. And we realise models as programs by programming real computers. So should we separate out problem solving, model making and programming, or are they inextricably linked? We also need languages and notations to solve problems, make models and write programs. Should these be the same or different? And isn't this all just Computing Science?

I think that it's important to distinguish *Computing* from Computing Science. Computing Science is an academic discipline which underpins all ICT, especially model making and tool making. Computing Science is concerned with the theory and practice of computations, which involves making models of reality from information structures and algorithms, and then animating the models on computers. That is, programming bridges models and computers.

It has been argued that everyone needs to learn how to program and that somehow programming is the 'new Latin' (e.g. Naughton, 2012). Shein (2014) offers a useful summary. I think we need to teach everyone how to *think*, in particular how to characterise a problem before realising a solution in some given hardware and software technology.

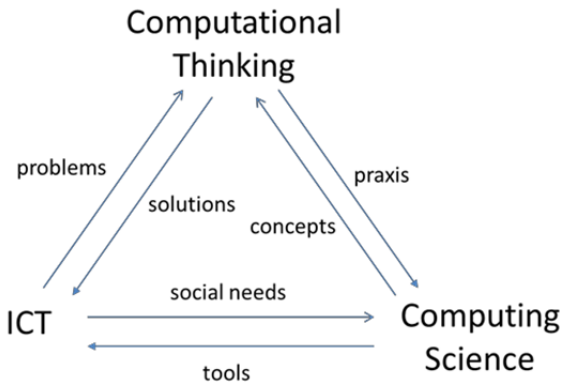

Fig 1: Computing triangle

I think we should view ICT, Computing Science and Computational Thinking, of which much more soon, as forming a Computing triangle – Figure 1 – where:

- Computational Thinking gives us a way of understanding problems;
- ICT offers problems to Computational Thinking in search of solutions;
- Computing Science:
  - Provides concepts for Computational Thinking in search of a praxis – that is synergy between theory and practice;
  - Responds to social needs from ICT with software tools, that is programs.

Thus, to solve problems, I think that we should stand apart from both ICT and Computing Science, and ask:

- How do we know when the problem is solved?
- What information is relevant to solving the problem?
- How must the information change for the problem to be solved?
- What computation(s) should we perform on the information to reach the solution?

In homage to Niklaus Wirth's insight that *Algorithms + Data Structures = Programs* (Wirth, 1973), given a problem, we might now say that:

Information + computations = solutions

Still, the hardest part of problem solving is characterising the problem.

2.2. Computational thinking and information

Since Wing's (2006) widely welcomed broadside, Computational Thinking (CT) has become seen as a key approach to problem solving. There are several different formulation of CT (Curzon, 2014) but Kao et al's (2011), strongly influenced by Wing, has wide currency. Here CT is based on four stages:

- 'Decomposition: the ability to break down a problem into subproblems.
- Pattern recognition: the ability to notice similarities, differences, properties, or trends in data.
- Pattern generalization: the ability to extract unnecessary details and generalize those that are necessary in order to define a concept or idea in general terms.
- Algorithm design: the ability to build a repeatable, step-by-step process to solve a particular problem.' (Kao, 2011)

While this formulation adds considerable precision to Wing, I think that it nonetheless presents a number of related difficulties. First of all it is vague, offering little guidance for how to apply computational thinking in solving concrete problems. Second, it neither distinguishes sufficiently between informational and computational aspects of CT, nor does it adequately link these in a unitary approach. Thirdly, the stages seem discrete, with little clear connectivity across them.

Thus, building on Kao et al, my formulation seeks to place information at the heart of CT and to link each stage to the next through the use of the interrogation of information to provide scaffolding for problem solving. I think that CT may be understood as:

1. *Decomposition* which is based on teasing out the basic building blocks of problems and involving:
   - identifying the information needed to solve the problem;
   - breaking the problem up into smaller sub-problems;
   - Identifying the sub-information needed to solve sub-problems.

2. *Pattern identification* which is based on finding differences and involving:
   - Looking for patterns amongst problems. This requires us to think about whether we've seen a problem like this before, and, if so, how the new problem is different;
   - Looking for patterns in the information. This requires us to consider how the information is structured, whether there are useful relationships within the information, whether we've seen information organised like this before, and, if so, how the new information is different.

3. *Pattern generalisation and abstraction* which are the inverse of pattern identification. They involve using the differences identified in pattern identification to:
   - Find the general cases for our problem. Here we think about what does and doesn't change in how the sub-problems are organised;
   - Clarify the general organisation of the information? Here we consider what information does and doesn't change in the overarching information structure.

That is, we find common templates for the things that don't change with slots for the things that do change.

4. *Algorithm design*, where we think about:
   - the sequence of steps from the initial information to the problem being solved;

- how the sub problems are connected;
- how the information changes between steps.

2.3. Problem solving techniques

2.3.1. Introduction

Now, it's easy to write down these stages but harder to see how they apply in practical problem solving for programming. It's really not clear where to begin.

Programming isn't hard when you know how to solve the problem. It then becomes a matter of battling with the vagaries of language-specific syntax, semantics and tools. For people new to programming, this language specific detail can become overwhelming, leading to a plethora of tiny, low level concerns at the expense of understanding how to solve an original problem.

And there is curiously little material on problem solving and programming. For example, Wienberg's (1971) classic study of the psychology of programming assumes that programming is an activity based on a specification that is elaborated from analysis, but says nothing about analysis itself. This echoes the then prevalent waterfall model of software development with distinct stages which are never revisited.

One of the few books that ostensibly focuses on problem solving and programming, Dromey's *How to solve it by computer* (Dromey, 1982), draws explicitly on Polya's 1950 foundational study *How to solve it* (Polya, 1990) and on later work by Wickelgren (1974).

Polya (pp5 & 6) characterises problem solving as a four stage process of:

- understanding the problem;
- linking unknowns to data to make a plan;
- carrying out the plan;
- looking back and reviewing the solution.

He offers a long list of problem solving heuristics, many of which correspond to different aspects of CT (e.g. analogy, auxiliary problem, decomposing and recombining, do you know a related problem, specialisation) but doubts any systematic way of deploying them:

> 'Rule of discovery. The first rule of discovery is to have brains and good luck. The second rule of discovery is to sit tight and wait till you get a bright idea. ... To find unfailing rules applicable to all sorts of problems is an old philosophical dream; but this dream will never be more than a dream.' (p172)

Nonetheless, Wickelgren attempts to provide a methodical approach to solving what he terms formal problems that is those couched in some formal notation, typically logical or mathematical. Wickelgren sees a problem as being specified as a starting state, a set of allowable operations over states, and a goal state. Thus, a solution is found by a sequence of state to state transitions leading from the start state to the goal state. Much of the book focuses on techniques for pruning the space of transitions, in particular reasoning backwards from the goal, but there is little on problem formulation.

Dromey (1982) is a proponent of top down design and stepwise refinement, linking decomposition to algorithm, which we will consider briefly below. He also uses logical statements to capture properties of program stages, typically loop invariants. While he acknowledges the central role of the choice of data structures in programming, he largely focuses on algorithm design, suggesting that structures are somehow chosen from a menu of

options. Despite acknowledging Polya's and Winkelgren's influences, Dromey has little to say about problem formulation.

Still, we already have tried and tested techniques for teaching programming so why can't we retrofit CT to what we do already? Let us now consider a range of these in slightly more detail, in inconsistently chronological order. Please note that much of the following is partial, anecdotal and superficial.

2.3.2. Programming language oriented

The oldest approach to programming is to just write down code, that is somehow using a programming language as some language of thought. Arguably, this programming language oriented approach is what the examples in Lovelace's 1842 account of Babbage's Analytic Engine represent (Morrison, 1961). Today we call this extremely widespread and long lived approach hacking. From a mathematician's perspective, using a programming language to construct algorithms is directly akin to using any mathematical notation to solve problems.

However, ad-hoc language oriented techniques suitable for solving small, well defined problems scale poorly for even moderately sized problems. Furthermore, effective hacking requires a deep facility with the programming language.

While language choice for teaching programming remains controversial, it is widely agreed that high level languages are more suitable for beginners than low level forms like assembly language and machine code. Nonetheless, programming languages in themselves do not offer any approach to problem solving beyond a means of formulating algorithms.

Some favour the use of an *industrial strength programming language*, for example FORTRAN, COBOL, C/C++, Ada and Java. Skill in an industrial language is seen as attractive to employers but such languages may be less well suited for conveying key principles to beginners. A more nuanced approach is to start with a *teaching subset* of a full strength language, leaving full detail to more advanced courses.

Others prefer to use a language originally intended for *educational purposes*, which often have sounder pedagogic bases than industrial languages. Some, for example Pascal, BASIC and the ALGOLs have crossed over into industrial use.

It is easy to overlook the use of *scripting languages* in teaching programming. Such languages, especially HTML, are widely used in early teaching of ICT skills for web design but, as they are used to layout and instantiate interface components, this may not be conceived of as programming.

Another alternative is to deploy a *language-neutral pseudocode*. This avoids the often observed problem of an excess focus on specific language syntax, and of first language bias, which both make it hard to generalise programming concepts for realisation in new languages (Cutts, 2014). Pseudocode works particularly well for presenting algorithms. For example, Knuth's (1968) classic text on fundamental algorithms uses both structured English and the MIX pseudo-assembly language. However, it is much harder to come up with a language-neutral notation for data structures, where extant languages show considerably more variation. For one approach, see Horowitz and Sahni's (Horowitz, 1976) structured pseudocode. Nonetheless, overall, I think that a pseudocode offers a valuable bridge from fundamental concepts to multiple languages.

2.3.3. Component oriented

Component based programming is based on using a programming language to compose pre-given elements, typically assembled in libraries. The components themselves represent

abstractions by domain experts and their use requires problem decomposition to identify appropriate components.

This approach also has a long history. For example, Wilkes (1951) describes the predominantly mathematical program library for the late 1940's EDSAC computer. Enormous libraries are available for contemporary programming languages, but, as in Wilkes' day, intimate understanding of library contents is crucial for effective component use. Indeed, an original problem decomposition may have to be modified, and additional algorithmic glue contrived, to make the problem specific components fit the library components.

Of considerable contemporary interest for initial teaching of programming are *component based languages* where learning is based on composing problem domain specific components. Thus, Logo (Papert, 1980) provides turtle graphics operations. More recent *visual languages*, like Alice, Scratch, BYOB, Snap and AppInventor, offer richer component sets, augmenting graphics with sound and animation. These are increasingly used in very early learning of programming, say at primary level. A disadvantage is that such languages take up more space than their textual equivalents, markedly limiting what can be presented. Once again, there is a tendency to revert to language oriented teaching.

2.3.4. Flowcharts

Diagrams have long been used in designing software and teaching programming. The use of *flowcharts* to design software is almost as old as digital computers, originating in von Neumann and Goldstine's reports from 1947 and 1948 (Haigh, 2014), and is still surprisingly wide today. At simplest, flowcharts are based on boxes for decisions and commands, linked by arcs indicating flow of control. As such, they combine the decomposition and algorithm CT stages. Flowcharting has long been an adjunct to learning a programming language. However, flowcharts are greedy of space, there is no clear discipline for their use, and, at worst, there may be one box for each command.

2.3.5. Structured programming

Early programming languages relied heavily on constructs oriented to the underlying platform, in particular the ability to branch to different instructions in memory, abstracted as GOTOs and labels in high level languages. Following Dijkstra's (1968) highly influential critique of branching, the use of *structured programming* (Dijkstra, 1969) was widely advocated, based on the use of the basic constructs of sequence, choice and repetition. Like flowcharting, this combines the decomposition and algorithm CT stages.

Structured programming was complemented by *modular programming* (Parnas, 1972), an approach where programs are composed from identified components. Parnas suggested that techniques for identifying modules are to decompose 'major steps', starting from flowcharts, and 'information hiding', where a module abstracts away from underlying detail, typically formed from a data structure and associated operations, prefiguring object-orientation. However, no methodology is offered for so doing.

Wirth's (1971) *stepwise refinement*, a systematisation of top down design for structured programming, was widely used as a teaching approach from the late 1970s. Here, a broad top level specification is refined to lower and lower levels, systematically identifying modules and structured programming constructs. However, refinement decisions tend to be driven by algorithmic considerations and based largely on criteria of efficiency and aesthetics. Subsequently, Wirth (1973) further emphasised the need to identify and deploy both standard and novel *data structures and algorithms*, but again offering no clear advice on how these may be selected.

In the same period, there were a number of techniques that augmented and extended modular structured programming, making strong use of diagrams, in particular Yourdon and Constantine's structured design with *structure charts* (Yourdon, 1979) and Constantine's *data flow diagrams* (Stevens, 1974). Where structure charts reflect a modular hierarchy of algorithmic components, data flow diagrams focus on the movement of information between algorithmic components. Once again, no method is offered for identifying the components in modular hierarchies or the information that flows between them.

In contrast, Jackson's (1975) structured programming (JSP) linked the structure of the data stream to the structure of the program that processed it. JSP was widely used as the methodology for teaching commercial data processing techniques in COBOL. However, JSP was only applicable to a highly constrained class of programs and gave no support for identifying intermediate information structures.

2.3.6. Declarative programming

Following the announcement of the Japanese 5th Generation Computer Systems Project in 1982 (McCorduck,1983), there was renewed pedagogic interest in the use of *declarative languages* for initial teaching of programming. Broadly, such languages are rooted in mathematical logic formalisms, with a distinction made between *functional languages*, based on lambda calculus and recursive function theory (Michaelson, 1989), for example Standard ML and Haskell, and *logic languages* based on predicate calculus, for example Prolog (Clocksin, 1981). Unlike other approaches, declarative programming strongly emphasises the CT techniques of abstraction, and pattern identification and generalisation, enabling a close link between algorithm and data structure. And polymorphic functional programming encouraged the complementary use of *types* and *higher order functions* for abstraction in structuring programs. However, the core declarative trope of recursion makes progress beyond elementary programming challenging, especially for those already used to procedural/structured approaches. Furthermore, declarative programming represented a reversion to a language oriented approach.

2.3.7. Object oriented programming

The leading contemporary approach to initial teaching of programming, at least in Higher Education, is *object orientated* (OO) where components (classes) encapsulate both data structures and algorithms (methods) to manipulate them. While OO languages have a long pedigree (e.g. Simula, Smalltalk), today Java predominates for teaching, followed by C++ and Python.

One popular contemporary approach to OO teaching is the components based *objects first* (Barnes,2012), where learners are given libraries of classes and are taught how to instantiate and compose them. This is reminiscent of Logo and visual languages mentioned above.

OO programming is complemented by the *Unified Modelling Language* (UML) (Stevens, 2000), a rich, multi-faceted diagrammatic approach, drawing on earlier structured approaches. UML is well suited to problem decomposition and identification of composable components for realisation as objects. However, it has little to say about information structures.

Design patterns (Gamma, 1995) are abstractions from common computations and offer standard templates that may be instantiated with problem specific components, providing a bridge from decomposition to algorithm. Design patterns are widely used for OO. Once again, considerable facility with the patterns is required for their effective use.

2.3.8. Parallel programming approaches

One domain where systematic problem solving is vital for efficient use of computational resources, and where problem taxonomies are widely used, is parallel programming. We will next briefly consider three methodologies for constructing parallel programs which have strong resonances with CT and which explicitly consider information decomposition.

Foster's approach (Foster, 1995) has partitioning and communication stages concerned with identifying tasks for subsequent optimal placement on processors in agglomeration and mapping stages. The partitioning phase involves information centric domain decomposition and computation centric functional decomposition, both to atomic components. Subsequently, these components are grouped to optimise allocation of tasks to available processors and to minimise inter-processor information communication.

Mattson et al's (2005) parallel design patterns strongly complement Foster's approach. As with Foster, the first stage, finding concurrency, involves task and information decomposition to atomic components. Subsequently, components are grouped using standard patterns for parallelism with known coordination and processing properties.

Finally, Cole's algorithmic skeletons (Cole, 1989) offer abstract constructs with well characterised properties for coordinating multi-processor computations. Skeletons may be provided in libraries and design patterns used to select them.

2.3.9. Problem based programming and problem based learning

It would be misleading to present the above approaches as somehow abstracted from the process of solving real problems. Indeed, the teaching of programming is perhaps unique in being so thoroughly grounded in, and driven by, seemingly endless open-ended concrete exemplars, often chosen to be motivational.

In such *problem based programming*, learners are typically encouraged to understand, modify and extend extant solutions to problems or sub-problems. Here, the emphasis is strongly on comprehension of the specific solution in context, not on either how the solution was derived or how it might be generalised to other solutions. The latter would require some taxonomy of problems.

It is interesting to contrast problem based programming with the application of *problem based learning* (PBL) to programming. PBL originated in medical education and is widely deployed in professional training. Nuutila et al (2005), who have applied PBL to teaching programming, cite Schmidt (Schmidt, 1983) in observing that:

> 'PBL refers to learning that is stimulated by descriptions of real world problems. Students do not necessarily try to solve a problem but rather define learning goals to better understand it.'(p125)

They distinguish PBL from *problem-oriented learning*, where 'learning takes place as a side effect of the solution process' and *project-based learning*, where 'the size of the problem becomes larger'. For example, Sierra et al (2013) explicitly conflate problem and project based learning in teaching programming. Here, the project is very precisely specified and sculptured to guide the learner to well defined outcomes.

I suspect that, as Nuutila et al suggest for two studies they cite, much teaching of programming that is claimed to be based on PBL is actually problem-oriented or project-based learning.

Furthermore, PBL must be fuelled by at least some ground knowledge. For example, Ma et al's (2005) approach combines pair programming with PBL. In what they call the initial '*explore* phase', students are already equipped with 'basic concepts, library classes and relevant algorithms'. Similarly, while O'Kelly et al's (2004) PBL-based approach uses stepwise refinement for problem decomposition, it builds on basic programming concepts such as 'looping, recursion, conditional statements' for solving 'problems of increasing difficulty'. Once again, in both cases there is some emphasis on the later stages of programming.

2.3.10. Summary

We have seen that each of the approaches corresponds to a subset of CT stages, with decomposition and algorithms predominating. The emphasis on algorithms is reflected in Curzon et al's (2014) contemporary account of CT.

Thus, in terms of which programming oriented approach to choose, really, all we can say is that fashions change. There are too many possibilities and none work easily beyond simple cases. More to the point, they all have a strong focus on the final program.

Nonetheless, in the following attempt to formulate an information-driven approach to problem solving through CT, the reader will spot the direct influence and opportunistic use of diverse aspects of the approaches discussed above.

3. Information driven computational thinking

3.1. Information structures

Our familiar approaches are certainly useful at the end of problem solving when we come to implement a solution, but at the start it's vital to focus on the problem without regard for the implementation. I think that the key is to work out how to characterise the information and computations. And it's best to start with the information, not the computation.

We can characterise information as:

information = base elements + structure

Note that we are not concerned with type or class, which are programming concepts to do with representing and implementing problem information.

Here, the base elements are the names of real-world things. At simplest, these are represented as sequences of characters to form meaningful unitary entities like words and numbers. Our things may also be built out of sub-elements, that is, things themselves may be structured.

Thus, we find:

- sequences: things before and after each other, either unordered or ordered on some property;
- tables: things arranged in rows of columns;
- arrays: things accessed by indices;
- records: things accessed by field name;
- lists: things arranged in chains, accessed by heads and tails;
- trees: things arranged in branching structures, accessed by branches;
- graphs: things arranged as nodes linked by arcs.

Note again that we are not concerned with data structures which are programming concepts to do with representing and implementing problem information structures.

Note also that structures have equivalences. Thus, a table is an array or records of row/column contents; an array is a list of index/value records; a list is an array of records of heads and tails. That is there are different, equally valid ways of characterising an information structure. Ultimately, the choice of structure is pragmatic: what gives the best characterisation of the problem in terms of, say, comprehension or abstraction.

3.2. Finding information structures

Alas, it's still no clearer where we should begin. It is tempting to go back to good old fashioned Computer Science. But there's no need. Rather, we should think about how real-world things are organised, and look at lots of concrete examples. Once we adopt an informational thinking approach, we can see information structures everywhere, often in how physical things are arranged but more conventionally in how data is organised.

For example, try thinking about how we interact with:

- parked cars;
- numbered houses;
- supermarket queues;
- English v Scottish bank queues;
- shopping lists;
- receipts;
- bills;
- account statements;
- itineraries;
- diaries;
- calendars;
- invitation lists;
- address books;
- seating diagrams;
- shop catalogues;
- library catalogues;
- family trees;
- cladistic trees;
- decision trees;
- underground maps;
- road maps;
- lottery tickets;
- betting slips;
- sports league tables;
- mobile contacts;
- browser favourites;
- social media friends;
- digital photo albums.

To characterise the information structure we need to interrogate the information. We need to ask how the information is organised, that is if it is:

- simple or composite;
- linear or grid or branching or cyclical;
- unordered or ordered;
- fixed or changeable in content or size or shape.

We can approach this by thinking about how to *access* the elements of the information structure by asking:

- why do we want to access the elements?
- which elements do we want to access?
- how do why/which affect access?
- where do we start the access?
- how do we continue access?
- how do we know if we've been successful or unsuccessful?
- what do we do if we're unsuccessful?

In the same way, we can ask how, if at all, we can *add* or *delete* or *modify* elements.

3.3. Generalising information structures

We now have some vague inkling about how to identify a concrete information structure that fits some specific real-world problem scenario. Applying computational thinking, we can now try to generalise by comparing apparently disparate information structures. Thus we can ask what stays the same and what changes. We can also look for similarities in concrete detail and in gross structure, ignoring the elements. In particular, are there commonalities in how to access, update, add and delete elements. Ultimately, we can use these comparisons to draw out the abstract structures we identified above.

For example, what do a car park and tables in a café, a bank statement and a utility bill, an allocated seat in an aircraft and in a cinema, a road map and an underground map, have in common? What are their differences? What common abstract information structure is suitable for both of them?

Finally, we can start to formalise the idea of an information structure as an *abstract data type*, with ways to:

- make a new structure;
- add information;
- check if information is present;
- find information;
- change information;
- remove information.

Not all information structures need have all these capabilities. For example, in many real world information structures, the information cannot easily be changed.

3.4. From information to computation

We can view these operations on abstract data types as being like the verbs of a language. We then make sentences by apply the verbs to nouns. That is, computation structures operations on information structures into algorithms. The key here is to make the structure of the computation follow the structure of the information. Note that we're making algorithms; we're not yet programming.

Once again, we should be driven by the problem scenario. We know that a computation solves a problem by turning old information into new information. So in the problem scenario we need to explore the sequences of information change.

Typically, we want to traverse the information structure, often visiting each element once, and stopping when some condition is met. On the way, we might do something to each element and accumulate some intermediate information.

For information arranged in fixed sequences, we typically want to traverse from some first element to some last element. So we need a notion of the next or current element. This is termed *bounded iteration*.

In other cases, we might want to continue traversal until some more general property is satisfied, and this might involve repeatedly visiting the same information or the same locations in the structure. This is termed *unbounded iteration*.

In both cases, we need to know how to start, continue and end the iteration, and this will be intimately associated with how we access and modify the information structure.

An alternative to iteration is *recursion*. Here, if we've got to the end of the structure, we stop traversal and maybe return a final value. Otherwise, we do something with the current element and then traverse the rest of the structure. Here we can distinguish the *base case*, where we stop, from the *recursion case* where we do the same thing to the rest of the structure.

As with iteration, we need to know how to start, continue and end the recursion, again closely following the information structure's properties.

Recursion and iteration are equivalent in expressive power. Alas, recursion is often seen as scary and advanced. In fact, in my experience, if you introduce recursion before iteration, students find it natural. One way is through children's counting songs like:

- Ten green bottles;
- On man went to mow;
- Twelve days of Christmas.

…where the counting structure lends itself naturally to thinking about the rest of the song in terms of how it follows from the current verse.

During information structure traversal, we often need to keep track of intermediate stages. We may wish to remember different positions in a structure, for example where we last found something or partial results, for example the value of the last element we found satisfying some property.

We can now introduce the idea of a variable as a general name/value association, where we can change the value associated with the name from a computation, often using the previous value.

I think that introducing variables should be delayed until they are needed to manage the stages of information structure traversal.

Coming back to accessing or changing an element in an information structure, we need to know how we can uniquely identify an element. Typically, we use what we might think of as a compound variable, for example the name of the structure qualified by a named field identifier or a numeric index.

Sometimes, we want to change an element regardless of its properties. But we may only want to change it if it satisfies some criteria. So now we can introduce notions of condition and choice. Similarly, we may want to change what we do next depending on properties of elements. That is, we can use conditions and choice to manage the stages of traversal.

Finally, we can stand back and think about whether the computation is necessarily:

- iterative, or could it be recursive?
- sequential, or could it be concurrent?
- linear, or could it be backtracking?
- deterministic, could it be non-deterministic?
- bounded, or could it be unbounded?

3.5. Notation

I have quite deliberately talked about problem solving in very general terms, without using any notation. But we do need to describe all these aspects of solving a specific problem in a concrete, consistent manner, so why don't we just use a programming language? After all, programming and seeing things working at an early stage are both highly motivating.

One problem is that the choice of language affects what can be described. Furthermore, the fine detail of a specific language can get in the way of understanding fundamental concepts. Finally, as noted above, the language-centric techniques used to knock up quick hit, small programs usually don't scale well to larger problems and this can prove frustrating and demotivating.

I think that, for problem solving, we should use a neutral notation like a pseudo code, for example the Haggis pseudo code (Michaelson, 2104).

4. Conclusions

To conclude, in trying to elaborate a pedagogy of programming, I think that we should be driven by problem solving. We've seen that:

Computational thinking =

Decomposition+ abstraction + patterns + algorithms

Now, CT is a framework not a recipe. In CT, the components overlap and interact. It just isn't possible to separate out definite stages of decomposition, abstraction, patterns and algorithms. Finding these is itself a creative, iterative activity.

I've argued that that classic CT overemphasises computation over information. For me:

solution = information + computation

I think that we should let the information structure the computation, and we should start with concrete instances of our problem scenario. We should then use CT to ask good questions, to tease out well known information structures and to guide computation design.

Finally, I think that, as with all education, we should view our students as active subjects and try to ground our practice in solving substantial problems drawn from their real world experiences, and expressed in terms they are likely to find familiar and engaging.

Acknowledgements

I'd like to thank:

- Quintin Cutts of the University of Glasgow, for prompting me to think about this peculiar stuff in the first place;
- Ian King of Kelso High School and his Scottish Borders teacher colleagues, with whom I first explored this material;
- participants of my workshop at the 2013 Computing At School Conference in Birmingham;

- Elaine Kao of Google Education, Simon Humphreys of Computing at School and Jeremy Scott of George Heriot's School, Edinburgh for discussion about the origins and definitions of CT;
- David Mathew of the *JPD* for his encouragement and support, and the anonymous referees for their irritating yet thought provoking comments;
- my long suffering students and colleagues.

References

(Barnes, 2012) D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction using BlueJ*, Fifth edition, Prentice Hall / Pearson Education, 2012.

(Clocksin, 1981) W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer Verlag 1981.

(Cole, 1989) M. Cole, *Algorithmic Skeletons: structured management of parallel computation*, Pitman, 1989

(Curzon, 2014) P. Curzon, M. Dorling, T. Ng, C. Selby and J. Woollard, Developing computational thinking in the classroom: a framework, Computing at School, 2014.

(Cutts 2014) Q. Cutts, R. Connor, P. Donaldson and G. Michaelson, `Code or (Not Code): Separating Formal and Natural Language in CS Education', WIPSCE 2014, 9th Workshop in Primary and Secondary Computing Education, Bonn, Germany, Nov 5-7, 2014

(Dijkstra, 1968) E. Dijkstra, Go To Statement Considered Harmful, *Communications of the ACM*, Vol. 11, No. 3, March, 1968, pp366-71,in E. N. Yourdon (Ed), *Classics in Software Engineering*, Yourdon Press, 1979.

(Dijkstra, 1969) E. Dijkstra, Structured Programming, in *Report of NATO Science Committee Conference*, Rome, Italy, October 1969, in E. N. Yourdon (Ed), *Classics in Software Engineering*, Yourdon Press, 1979.

(Dromey, 1982) R.G. Dromey, *How to Solve it by Computer,* Prentice-Hall, 1982.

(Foster, 1995) Ian Foster, *Designing & Building Parallel Programs: Concepts & Tools for Parallel Software Engineering*, Addison-Wesley, 1995.

(Gamma, 1995) E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

(Haigh, 2014) T. Haigh, M. Priestly and C. Rope, Los Alamos Bets on ENIAC: Nuclear Monte Carlo Simulations, 1947-48, *IEE Annals of the History of Computing*, Vol 36, No. 3, July-Sept 2014, pp42-63.

(Horowitz, 1976) E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Pitman, 1976.

(Jackson, 1975) M. A. Jackson, *Principles of Program Design*, Academic Press, 1975.

(Kao, 2011) E. Kao, Exploring Computational Thinking at Google, *CSTA Voice*, Vol 7, Issue 2, May, 2011, p6.
http://csta.acm.org/Communications/subCSTAVoice_Files/csta_voice_files/csta_voice_05_2011.pdf

(Knuth, 1968) D. Knuth, *Fundamental Algorithms: The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1968.

(Ma, 2005) L. Ma, J. D. Ferguson, M. Roper, and M. Wood, A collaborative approach to learning programming: a hybrid learning model. In: *6th Annual Higher Education Academy Subject Network for Information Computer Science conference*, August, 2005.

(Mattson, 2005) T. Mattson, B. Sanders and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005

(McCorduck, 1983) P. McCorduck, *Introduction to the Fifth Generation*, Communications of the ACM, Vol. 26, Number 9, pp629-30, September, 1983.

(Michaelson, 1989) *An Introduction to Functional Programming Through Lambda Calculus*, Addison-Wesley, 1979.

(Michaelson, 2014) G. Michaelson and Q. Cutts, *Haggis: full pseudocode specification*, http://www.macs.hw.ac.uk/~greg/Teaching%20Computing/Haggis%202.docx [accessed 2/5/14]

(Morrison, 1961) A. A. Lovelace, Sketch of the Analytic Engine Invented by Charles Babbage by L. F. Menabrea. With Notes upon the Memoir by the Translator, Ada Augusta Countess of Lovelace, in P. Morrison and E,. Morrison (Eds), *Charles Babbage and his Calculating Engines*, Dover, 1961.

(Naughton, 2012) J. Naughton, Why all our kids should be taught how to code, *Observer New Review*, 31/3/12, http://www.theguardian.com/education/2012/mar/31/why-kids-should-be-taught-code [accessed 2/5/14]

(Nuutila, 2005) E. Nuutila, S. Törmä and L. Malmi, PBL and Computer programming – the Seven Steps Method with Adaptations, *Computer Science Education*, Vol. 15, No. 2, pp123-142, June, 2005.

(O'Kelly, 2005) J. O'Kelly, J. A. Mooney, S. Bergin, S.Dunne, P. Gaughran, and J.Ghent, An Overview of the Integration of Problem Based Learning into an exiting Computer Science Programming Module, *Pleasure by Learning*, Cancun, Mexico, 2004.

(Papert, 1980) S. Papert, Computer-based microworlds as incubators for powerful ideas. In R. Taylor (Ed.), The computer in the school: Tutor, tool, tutee (pp. 203–210). New York: Teacher's College Press, 1980.

(Parnas, 1972) D. Parnas, On the Criteria to Be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol 5, No. 12, pp1053-58, December, 1972, in E. N. Yourdon (Ed), *Classics in Software Engineering*, Yourdon Press, 1979.

(Polya, 1990) G. Polya, *How to Solve It (2nd Edition)*, Penguin, 1990.

(Schmidt, 1983) H. Schmidt, Problem-based learning: rational and description, *Medical Education*, Vol. 17, pp11-16,

1983.
(Shein, 2014) E. Shein, Should *Everybody* Learn to Code?, CACM, Vol 57, No 2, February 2014, p16-17.
(Sierra, 2013) A. J. Sierra, T. Ariza and F. J. Fernandez, PBL in Programming Subjects at Engineering, *Bulletin of the IEEE Technical Committee on Learning Technology*, Vol. 15, No. 2, April, 2013.
(Stevens, 1974) W. Stevens, G. Myers and L. Constantine, Structured Design, IBM Systems Journal, Vol. 13, No. 2, May , 1974, pp115-39, in E. N. Yourdon (Ed), *Classics in Software Engineering*, Yourdon Press, 1979.
(Stevens, 2000) P. Stevens with R. Pooley, *Using UML: Software Engineering with Objects and Components*, Addison-Wesley, 2000.
(Wickelgren,1974). W.A. Wickelgren, *How to Solve Problems*, Freeman, 1974.
(Yourdon, 1979) E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design,* Prentice-Hall, 1979.
(Weinberg, 1971) G. M. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.
(Wilkes, 1951) M. Wilkes, *The preparation of programs for an electronic digital computer*, Addison-Wesley, 1951.
(Wing, 2006) J.M. Wing, Computational Thinking, *CACM Viewpoint*, March 2006, pp. 33-35.
(Wirth, 1971) N. Wirth Program Development by Stepwise Refinement, *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227
(Wirth,1973) N. Wirth, *Algorithms + Data Structures = Programs,* Prentice Hall, 1973.

Raising Awareness of Diversity and Social (In)justice Issues in Undergraduate Research Writing: Understanding Students and their Lives via Connecting Teaching and Research

Gloria Park, English Department, Indiana University of Pennsylvania

Abstract

Inspired by my own experiences as an undergraduate writing student who did not see a connection between my life and the topics of the courses, this article details my first ventures into designing and teaching sections of a research writing class, entitled *Researching Writing: Raising Awareness of Diversity and Social Justice Issues within and Beyond our Lives*. The purpose of this course was to promote issues of diversity and social (in)justice in a required liberal studies course. Interview data from undergraduate women students who participated in this research writing course from 2009-2011 were explored in order to uncover their experiences in the class and understand what they found effective or ineffective. The findings indicated that most of the students appreciated being able to choose their own research topic, and also found chunking parts of the research project more effective for understanding the research process. Although engaging students in research and course activities related to controversial issues is difficult, there is a need for more liberal studies courses to incorporate topics related to diversity and social (in)justice.

Key Words: Social (in)justice, diversity, research writing, liberal studies, curriculum

How it All Began

Beginning in fall 2008, my first semester as an assistant professor in the English Department housed in a university located in Western Pennsylvania, I was assigned three sections of Undergraduate Research Writing. Although I did not know everything I needed to know about how Undergraduate Liberal Studies Program courses (such as writing, literature, and other English department courses) were positioned in the department and the university, I was excited to teach my first undergraduate writing course.

As a way to put my *Self* into *Pedagogy,* I reflected on how writing, specifically research writing, in different points of my educational journey, has influenced me as a learner, teacher, and teacher educator. I went back as far as my undergraduate years at a university in Boston. As a pre-med student, my focus was to get through writing requirements in different disciplines. I

never really focused on how a writing course in my undergraduate years would impact the